Master Degree Thesis

Optimization of CNN-Based Object Detection Algorithms for Embedded Systems

Daniele Caschili
Matricola: 243807

Supervisors
prof. Massimo Poncino
doc. Daniele Jahier Pagliari

Master degree course in Ingegneria Informatica
(Computer Engineering)

Politecnico di Torino
Italia, Torino
Academic year 2018-2019

Abstract

Object detection is the branch of computer vision and machine learning that classifies and localizes multiple objects of different classes in the same image or a video stream. At this time, the state-of-the-art method in this field is to use a Convolutional Neural Network (CNN) to process the input images and produce boxes that enclose each detected object. These networks are highly accurate, but they require lots of computational power. Therefore, to run object detection algorithms on embedded devices the easiest solution is to connect to the cloud. This leads inevitably to latency and energy consumption problems. The optimization and reduction in the size of these networks is currently an open research problem.
This thesis proposes a technique to reduce the size and the computational requirements of CNN-based object detection by using post-training low bit-width quantization. Though reducing bit-width greatly reduces the size and computational requirements, it also drastically decrease the network prediction capabilities. Therefore, this thesis analyzes the possibility of using two consecutive inferences to obtain a highly-accurate result yet with an energy-efficient procedure. First, a reduced bit-width inference is used to roughly detect objects in the image. Then, these boxes are refined with a higher precision inference.
Results show that the refinement is able to increase the mAP value by about 10% with respect to the reduced bit-width network while having a similar energy consumption. The full precision network has greater precision, but it consumes twice the energy.

Contents

List of Figures

List of Tables

Chapter 1

Introduction, Motivations and Goals

The increase in computational power in last years allows employing machine learning techniques in various fields, from computer vision to speech recognition, etc. Whereas in the past, the main goal was to reach a certain computational power in order to make these machine learning algorithms work, nowadays the idea is to improve efficiency to move the inference phase from servers to embedded devices, limited both in battery life and speed. While the training phase can be performed without problems on the servers, the inference phase is better suited to be performed locally, avoiding all problems derived from a permanent network connection to a cluster, like latency, security, and energy consumption.
In literature, there are two paths to neural networks optimization: a hardware oriented approach and an algorithmic one [30]. The former generally proposes custom hardware accelerators to improve efficiency [24, 18]. The latter is additionally divided into a static approach and a dynamic one, where the static solutions try to improve the network efficiency independently of the input data [4, 8, 19] while the dynamic ones optimize different inputs in different ways[11, 21].
This thesis is focused mainly on Convolutional Neural Networks. The main power consuming process present in these networks is certainly the convolution operation which consists in a large number of MAC (multiply and accumulate) operations. In literature, the most commonly used method to reduce the complexity of Convolution operations is the bit width reduction via a quantization process: a mapping from a higher precision space to a lower bit width representation. By reducing the number of bits requested for those operations, it is possible to greatly reduce the energy consumption while keeping an acceptable accuracy score. However, most convolution approaches belong to the static category, in the sense that the bit-width is decided independently of input data.
Dynamic quantization techniques like [21, 11] adapt the optimization to the runtime conditions.
The main goal of this thesis work is to explore a possible dynamic approach to reduce the energy requirements of a Convolutional Neural Network for object detection. In summary, the method works as follows. A low energy cost network produces general and poorly defined predictions. Then, if the prediction score doesn't meet the requirements, a more expensive network is used to refine the result, applied on a section of the original image containing the bounding box predicted by the first

network. The expensive network can then operate using smaller size convolutions with less MAC, and can focus his analysis on a smaller area.
To verify the efficiency of this approach, we simulated the effect of uniform quantization. The two networks are modified using a custom network node, written in Tensorflow, that reproduces the quantization effects. This allows us to run the tests on a server reproducing the precision loss derived from bit-width reduced operations on an embedded system. To analyze the energy we used the results from **Moons et al.**[18]. They calculated the power consumption for MAC operations performed at 8 and 4 bits.
The proposed method is tested against a realistic industrial dataset, generated in the context of a European project, which contains images of objects moving on a conveyor belt, which have to be detected in order to perform automatic counting. In the dataset, objects are small and few per frame, which are the most favorable conditions for the applications of our method. In fact, the second inference step is performed on small regions of the original image, yielding a reduced overhead in terms of the number of operations. Without this condition, the refinement with the higher precision network would increase the overall process energy consumption, almost exceeding that of a solution that performs only the high precision analysis.
The process is performed on three different datasets: images with at most one object, images with at most two objects and the whole validation set.
We expect to find the best results using the one-object dataset but we have to consider the possibility to have some false positives. If the low precision network detects more than one box, even if we are using the one-object dataset, the high precision network will perform the refinement on all predictions with a low confidence score. So there is the possibility to waste energy in detecting the false positive produced by the low precision network.
The results show that, on all datasets, the refinement technique is able to improve by about 10% the mean Average Precision (mAP) score, which is the standard accuracy score for object detection, while keeping the energy cost similar to a solution that uses only a lower precision network.

Chapter 2

Background

Thanks to the increase in computer computational power and the availability of huge datasets, machine learning has become increasingly popular in the last years [31]. Machine learning is a branch of Artificial intelligence that aims at building programs able to learn to perform some actions through a process called training. Deep learning is a branch of machine learning that tries to build a deep neural network to achieve tasks in different fields such as object detection, speech recognition, self-driving cars, etc. [30]

2.1 Neural Networks

Inspired by the human brain with its billions of connections, deep neural networks have multiple layers of interconnected units called *neurons*. Depending on the signals it receives as inputs, the neuron can be activated, producing another signal sent to another neuron. The set of input signals are propagated through the middle layers, called *hidden layers*, and then to the output layer.

2.1.1 Neurons

Neurons are the building blocks of neural networks.

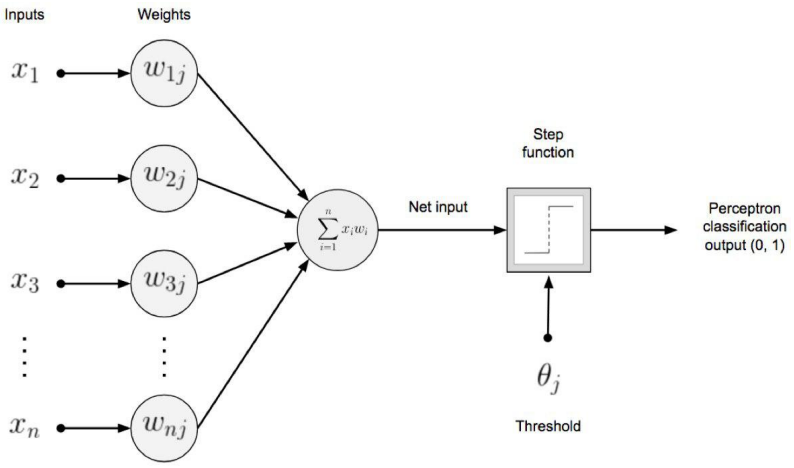
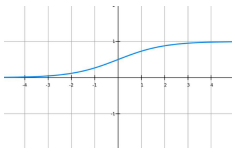


Figure 2.1: Neuron structure [22]

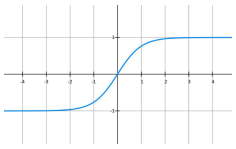
Figure 2.1 shows one of the most basic neuron structures. A set of input values are weighted and summed. This result is used as input for an activation function that determines how much the neuron will be activated by the received input signal. For this reason, the analogy with the human brain is strong, a biological neuron is activated by some input signal and propagates its output to others. In the image:

- x_i: is one of the input values.
- w_i: is one of the weights.
- θ: is the activation threshold.

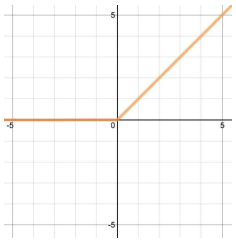
The activation function is just a "rule" to determine how much will be activated. There are different types of activation functions and the most common types are:



(a) Sigmoid activation function



(b) tanh activation function



(c) Relu activation function

Figure 2.2: Activation functions [22]

The function in figure 2.2a is called **sigmoid activation function** and produces an output in the range $[0, 1]$. It has been reported that the networks that use this activation function may incur in vanishing or exploding gradient problem [33]. The math formula for figure 2.2 is:

$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

The **hyperbolic activation function** (figure 2.2b) is similar to the sigmoid function, but its output range is $[-1, 1]$. Both tanh and sigmoid nonlinearities are defined as *saturated* due to the fact that they have a limited range of possible values. For that reason the usage of both tanh and sigmoid activation functions in feed forward neural network as hidden layer leads to vanishing gradient problems[7].

$$\tanh(x) = \frac{2}{1 + e^{-2x}} - 1$$

The last one, figure 2.2c, is the **ReLU** (Rectified Linear Unit) which offers a valid alternative to the previous functions, solving the problem of vanishing or exploding gradient [16].

$$\max(0, x)$$

Networks with many layers using this activation, usually learn faster [15]. The ReLU can lead to have some units that are never activated during training, due to the 0 part of the function [16]. A possible variation the *ReLU*, called **Leaky ReLU**, ensures that, even if x is negative, the activation function isn't zero. The formula is:

$$\max(0.01x, x)$$

2.1.2 Networks

A neural network is composed of multiple neurons organized in layers. The most basic structure of a neural network is the *feed-forward neural network*. Figure 2.3 shows a simple example:

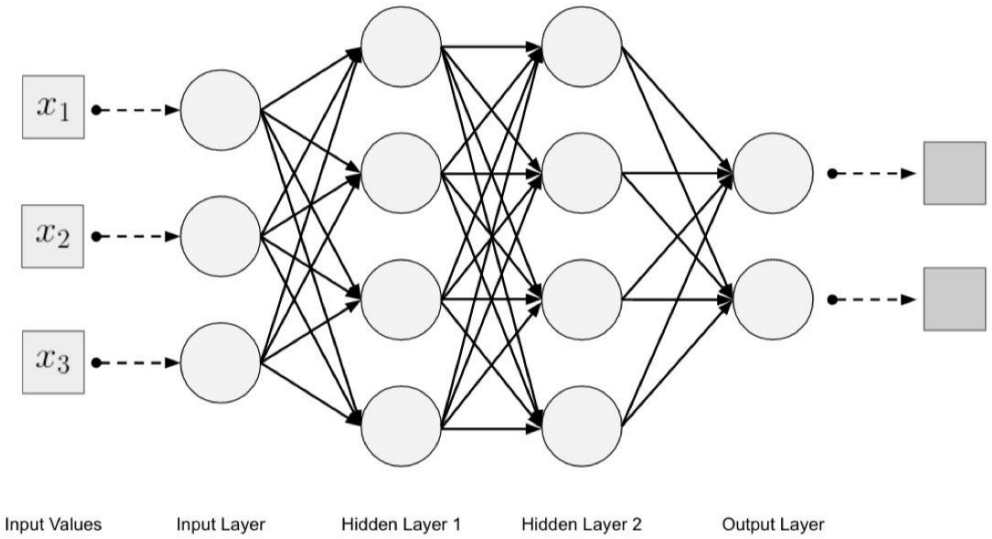


Figure 2.3: feed forward neural network [22]

This type of network uses the so called *fully-connected layers* that forward their output to all neurons in the following layer. Each internal layer is called *hidden*

layer. The *output layer* produces a set of probabilities values for a set of mutually exclusive classes. Generally, the last layer is a *softmax* layer, it converts the previous layer output to this probability distribution.

2.1.3 Training

Supervised learning is the most common form of training [15]. The training is performed using a set of input vectors paired with the corresponding desired output. Conceptually the network tries to mimic the training set. Practically, the network parameters are modified to reduce the value of a function that expresses the difference between the actual output and the ground truth, called *cost function* (also loss or objective function).
The training process uses a method called backpropagation to update the network weights.
Considering a single neuron, a modification in its weights directly modifies the output. If this neuron is connected to another one, the modification also affects the connected neuron. To take into account this sequence, the chain rule of calculus is used. Considering a function $h(x) = f(g(x))$ dependent on the output of another function, we can calculate the change in h due to a change in the input x using:

$$\partial h(x) = \frac{\partial f}{\partial g}\frac{\partial g}{\partial x}\partial x$$

This formula during backpropagation has to be applied to a longer sequence of interdependent functions.
Due to this complexity is not possible to calulate the exact combination to perform the cost function minimization. For that reason *gradient descent* is used.

$$C = \frac{1}{2n}\sum_{x}(y(x) - a)^2$$

Considering

- n: the number of inputs.
- $y(x)$: the desired output.
- a: is the actual output, clearly it also depends on both weights and biases.

This cost function is called *mean squared error*. Different Loss functions exist and each one modifies the performance of the network.
The training phase wants to reduce the loss by modifying the network parameters, weights, and biases. As mentioned above, given the complexity of the problem, the exact solution can't be found analytically. The gradient descent process modifies all parameters following the negative gradient of the cost function. To understand it, let us consider the case of C dependent only on 2 variables $C(w, b)$. A small change in those variable produces:

$$\Delta C(w, b) \approx \frac{\partial C}{\partial w}\Delta w + \frac{\partial C}{\partial b}\Delta b$$

Now let us name:

$$\nabla C = \left(\frac{\partial C}{\partial w}, \frac{\partial C}{\partial b}\right)^T$$

$$\Delta v = (\Delta w, \Delta b)^T$$

Where ∇C is called *gradient*.

$$\Delta C(w, b) \approx \dot{\nabla} C \Delta v$$

We need to decrease C so ΔC has to be negative. Hence we can chose $\Delta v = -\eta \nabla C$. In this way it is guaranteed that ΔC will be negative. $-\eta$ is called the *learning rate*. Then the modification is applied to the values

$$w \to w' = w - \eta \frac{\partial C}{\partial w}$$
$$b \to b' = b - \eta \frac{\partial C}{\partial b}$$

In the case of a real network, we have multiple weights and biases, so a partial derivative of the cost function has to be calculated using the chain rule explained above. But the base concept remains the same: we want to follow the gradient of C in the negative direction.
The whole gradient descent process would require the complete dataset. Due to the computational complexity, in general it is used the *stochastic gradient descent* over a *mini batch*. A small set of randomly selected input values is used as input in the network. The gradient descent is calculated for each input and the mean result is kept to update parameters. Then a new mini-batch is selected until all input values have been used. This is called a *epoch*. To complete the training process various epochs are performed.

2.2 Convolutional Neural Network

Convolutional neural networks are well suited to tasks such as object recognition, image classification, and text analysis. They have been first introduced in 1989 [13], but only in recent years, thanks to the increase in GPU power and to the availability of huge datasets for training, have been largely used in computer vision tasks. The key observation is that many natural signals are a composition of low-level features (edges form motifs, motifs assembles into parts, parts form objects [15]). These patterns can be found not only in images but also in speech and music.
The CNN structure is inspired by the visual cortex in animals [22], where groups of cells are sensitive to a small subregion of the input image. Therefore the image is not processed as a single block but as a composition of smaller features.
The main difference with *feed-forward neural networks* is the absence of fully connected layers in the initial and central part of the architecture. Fully connected

layers are only used in the final part to produce the output, the classification probability distribution. From a computational perspective, this translates into models that require a smaller number of weights even for a large number of layers, and are therefore more tractable from the point of view of memory occupation. The training process is executed with backpropagation as in the feed-forward neural network [15].

2.2.1 Architecture

The figure 2.4 shows an high-level view of a CNN organization. Apart from the input layer, the middle layers achieve feature extraction while the final fully connected part performs classification.

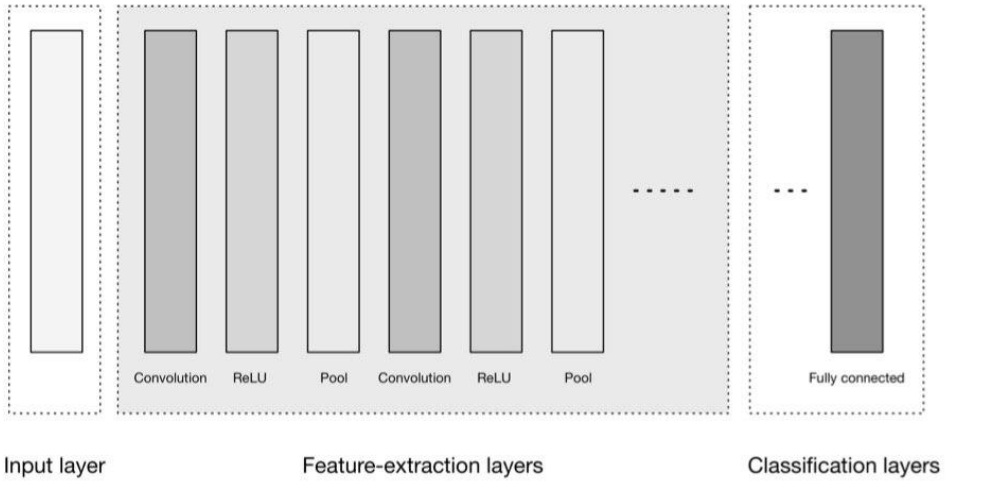


Figure 2.4: High level overview of CNN structures [22].

In basic CNN architectures, feature extraction is performed by a repeated pattern. First, a convolutional layer is applied to the input, then an activation function (generally the ReLU) and finally a Pooling layer, which reduces the information size. An example of one of the first CNN created is shown in figure 2.5.

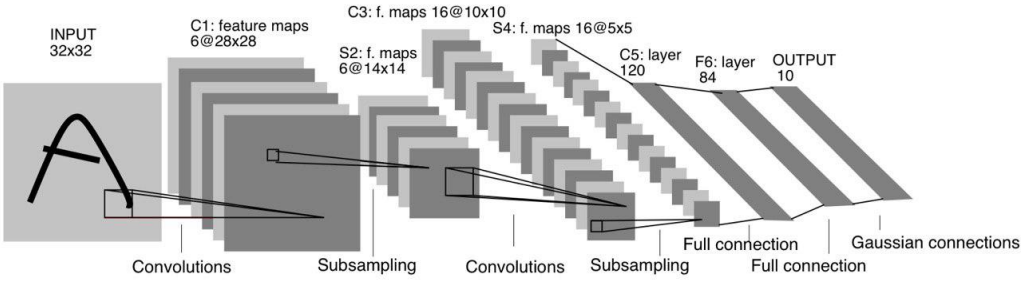


Figure 2.5: LeNet-5 architecture [14]

2.2.2 Layers

Convolutional layers' role is to find local conjunction of features from the previous layers [15]. They don't perform a simple matrix multiplication as the fully connected

in a feed-forward neural network do, they rather execute a convolution.
Weights in a convolutional neural network are grouped in matrices called *kernels* (or *filters*). Though they have a smaller width and height with respect to the input, in basic CNN architectures they must match the depth.
Considering the input layer, an image usually has three dimensions: width, height, and depth (that is generally three, following RGB encoding). Therefore the first set of filters must have a depth of three.

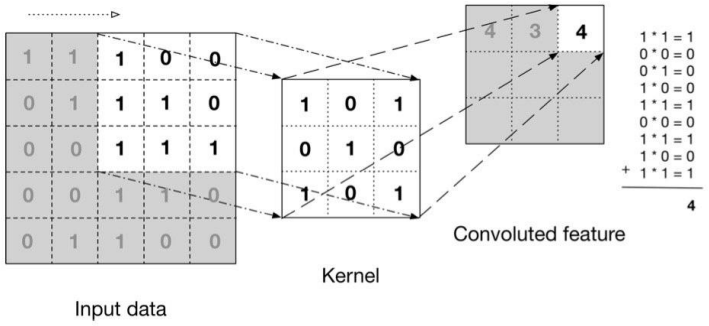


Figure 2.6: Convolution [22]

As shown in figure 2.6 the convolution operation is applied by multiplying a kernel for an area in the input image (each value is multiplied by the corresponding weight, then the result is the sum of all multiplications). The result is stored in the output matrix called **feature map** or **activation map**. There are multiple of such filters. Once the input has been completely processed, the network uses the next filter. It is important to notice, that the depth of the activation map is not related to the input or the kernel depth, instead, it is equal to the number of kernels applied to the input image.
Some definitions for this layer:

- N: width/height of the input, in the simplified case of a square input image.
- P: the amount of padding used in the input. The padding is useful to obtain a desired dimension in the activation map, for example, to keep the same input sizes.
- S: the stride. It expresses by how much a kernel is shifted during convolution.
- F: the kernel dimension.

Then we can express the size of the output feature map as:

$$\left\lfloor \frac{N + 2P - F}{S} \right\rfloor + 1$$

Pooling layers merge semantically similar features found in the previous activation map [15] and help control overfitting [22]. The most common layer is the **Max-pooling** which extract the maximum value.

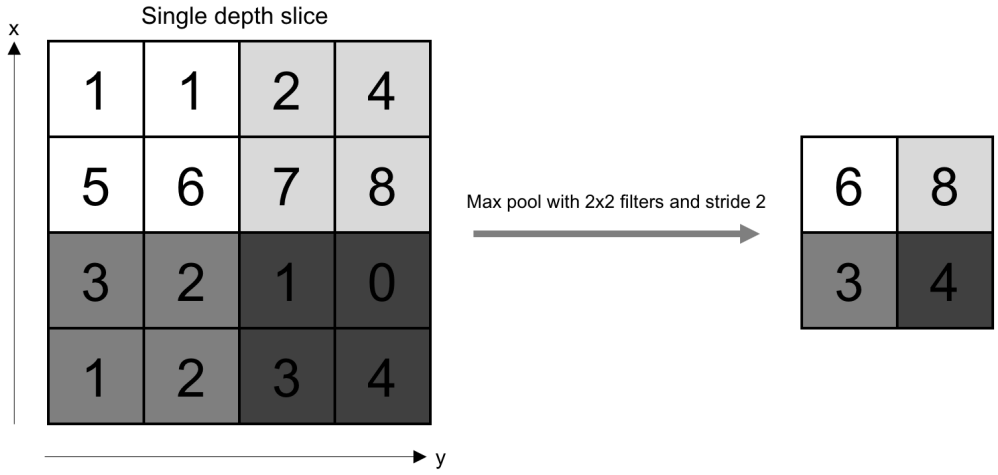


Figure 2.7: Max Pooling example [28]

The different colors highlight different areas of the input tensor to the polling layer. The maxpooling takes the maximum number in those sections. The stride is typically of two.

Activation layers apply an activation function to each element of their input (typically a ReLU or LeakyReLY).
The **fully connected** layer takes the output of convolution/pooling, flattens it and predicts the best label to describe the image. As in a normal feed-forward neural network, the inputs to the fully connected layer are multiplied by the weights and summed together. Then an activation function is used to produce the output. The results are propagated to the next fully connected layer. The last one has a neuron for each class label and it produces the probability distribution.

2.2.3 Training

The **training** phase modifies the weights values in order to detect image features. As we can see from figure 2.8, filters after training have learnt some low-level features of the image.

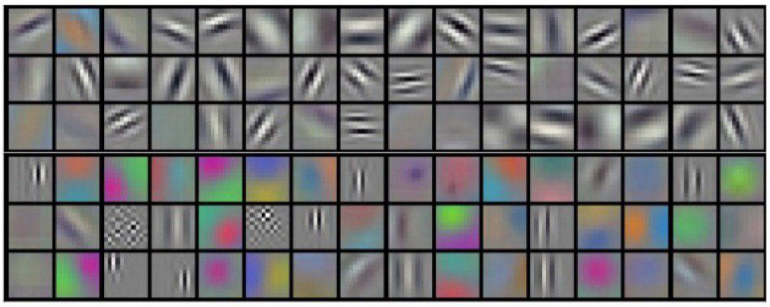


Figure 2.8: Filters learnt. [12]

Dropout technique is used to reduce overfitting during training [12]. It sets

to zero the output of each hidden neuron with a given probability reducing the co-adaptation of neurons.

Another technique called **batch normalization** limits the effects of *covariate shift*. The continuous change in the distribution of network activations due to the changes in network parameters forces layers to continuously adapt, slowing down training [9]. The solution is to apply, after activation layers, a normalization to bring the mean to zero and the standard deviation to one.

2.3 Object Detection

Object detection performs classification and localization even for images with multiple objects with different classes. The localization task is achieved drawing boxes around objects, generally defined in terms of box center, width, and height. Within each box, the object is also classified.

2.3.1 YOLO (You Only Look Once)

Networks like YOLO (You Only Look Once)[27] use a carefully built network to analyze the image in a single inference. YOLO can perform object detection on a video stream (45 frames per second according to the results of [27]). The input image is divided into an $S * S$ grid and for each cell, the network predicts a specified number of bounding boxes and outputs the conditional class probabilities. The number of outputs for YOLO are:

$$S * S * (B * 5 + C)$$

B is the number of boxes to be detected by each cell, this is a network parameter. C is the set of class conditional probability, these values are calculated for each cell for each class. The five numbers define a single bounding box:

- b_x, b_y the bounding box center.
- b_w, b_h the box height and width.
- BC the box confidence score.

where the box confidence score represents the probability of detecting an object and how good is its shape.

Even if more than one box is detected for each cell, at the end only one object per cell can be identified.

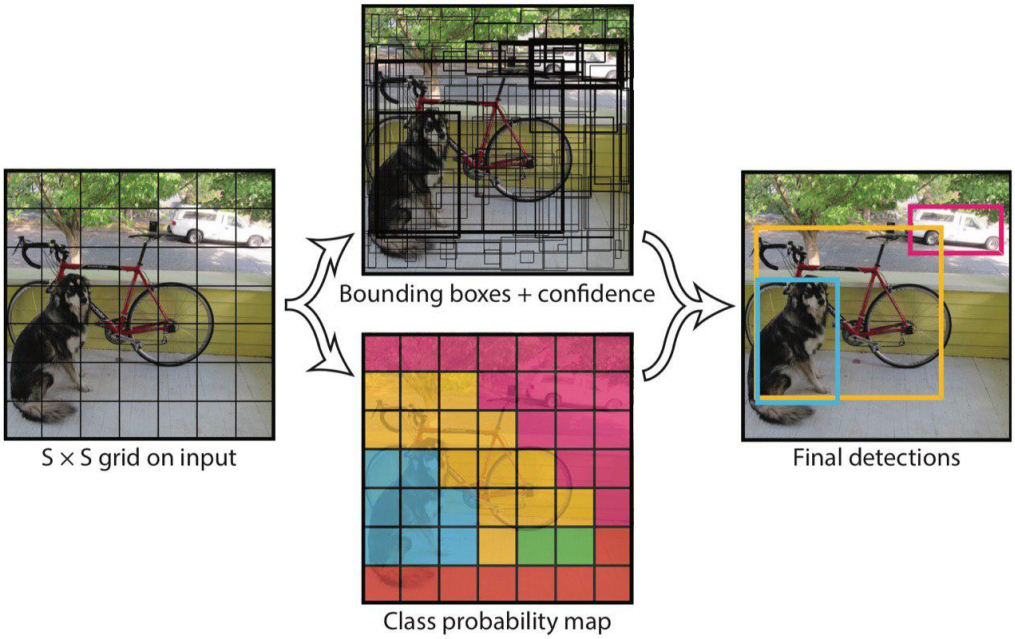


Figure 2.9: YOLO Model. The image is divided into $S * S$ cells and for each cell a certain number of bounding boxes are detected.

As we can see from figure 2.9 there are lots of boxes predictions, but only the best ones are kept in the final output. The metric to evaluate a box is the **IoU** (Intersection over Union):

$$IOU = \frac{\text{area of overlap}}{\text{area of union}} =$$

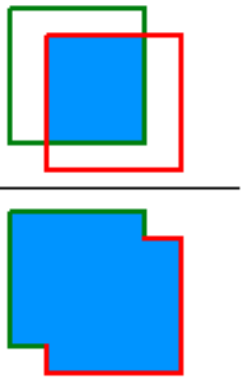


Figure 2.10: Intersection over union is the metric used to compare bounding boxes. It is the ratio between the overlapping area and the box union area [23].

In order to select the best box for each grid cell the **non-max suppression** algorithm is used [27]. First, all boxes with a confidence score below a certain threshold are removed, then the box with the highest probability is used to compute the IoU against all other. If the resulting IoU is greater than another threshold (usually 0.6), the compared box is discarded. Conceptually it means that they are boxes encircling the same object, given that they are almost completely overlapped. Given that the compared box has a lower confidence score, it is discarded.

2.3.2 YOLO v2

Version 2 of the YOLO network introduces various changes to improve speed (while keeping a good accuracy score of 67 frames per second [25]).
The network modifies the input size to accept $416 * 416$ images. In this way, the image center is overlapped to that of the central grid cell. It is common to have pictures with centered subjects so it simplifies certain predictions.
YOLO v2 uses *anchor boxes* to predict all the B boxes.
The box is represented using different values. The prediction values:

$$
\begin{aligned}
b_x &= \sigma(t_x) + c_x \\
b_y &= \sigma(t_y) + c_y \\
b_w &= p_w e^t_w \\
b_h &= p_h e^t_h \\
\sigma(t_0) &= Pr(object)
\end{aligned}
$$

Where $Pr(object)$ is the probability that the box contains an object. The $\sigma()$ (sigmoid) function is used to constrain values between $[0, 1]$. The box center is expressed as an offset with respect to the top left corner of the grid cell (c_x, c_y) and width and height are expressed relative to the anchor box dimensions (p_w is the anchor box width, p_h the height).
The value $\sigma(t_0)$ gives an idea of how good is a box, it is the box confidence score. If this value is below a certain threshold (typically 0.3) the corresponding box is discarded. Figure 2.11 shows the values of a predicted box. The dashed box is the anchor box while the blue box is the prediction. The blue dot is the predicted box center, it is expressed as an offset with respect to the top left corner of the cell in which it is contained plus the coordinates of that corner.

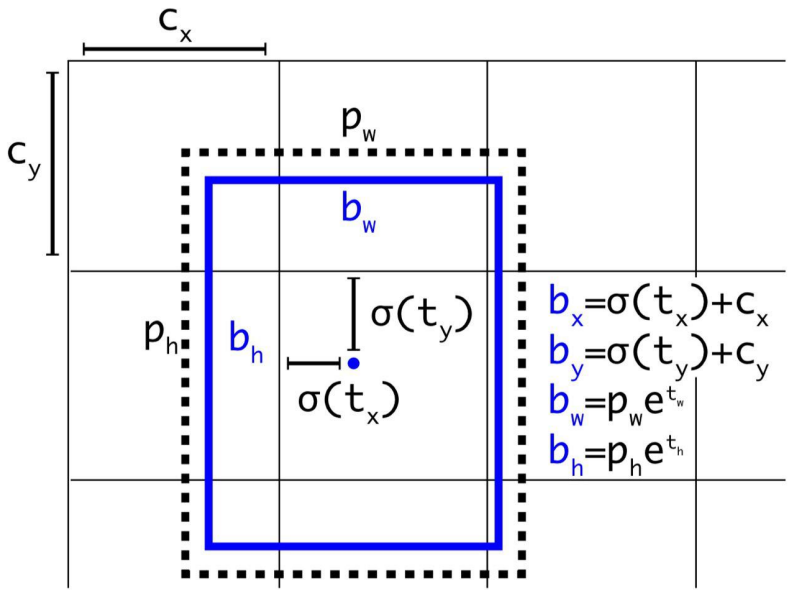


Figure 2.11: Box definition diagram [25]

The network architecture is similar to YOLO v1, but it doesn't have the last two fully connected layers.
The output has a different size:

$$S * S * (B * (5 + C))$$

Each predicted box has its conditional class probabilities, differently with respect to YOLO V1 that had a set of conditional class probabilities for each cell.

YOLO v3 The last version of YOLO, version 3 [26], brings some small changes and it is a little bit bigger, but produces fast and good result with respect to for example SSD (Single shot detection). The main change is that it predicts boxes at multiple scales.

2.3.3 Object detection evaluation - mAP

In order to evaluate how good a model is performing object detection, the mAP (mean average precision) is the most used metric.
The definition of the mAP depends on two other metrics:

$$Precision = \frac{TP}{TP + FP} = \frac{TP}{\text{All detections}}$$

$$Recall = \frac{TP}{TP + FN} = \frac{TP}{\text{All ground truths}}$$

where:

- TP: True Positive. Boxes with IoU > threshold.
- FP: False Positive. Boxes with IoU < threshold. Also boxes with IoU > threshold for a ground truth for which a prediction has already been found.
- FN: False Negative. The network didn't produce any boxes while a ground-truth exists.

The process starts by calculating the AP for each class, then the mean produces the mAP.
To calculate AP, the inference is performed on a set of images and the results are compared with the ground truth. Having the list of all predicted boxes, it is possible to start calculating the TP and FP. If more than one box has an IoU greater than the threshold with the same ground truth, only the box with the highest IoU is considered TP, the others FP.
The list of boxes is ordered based on the confidence score, in decreasing order. Then,

starting from the highest confidence score, the precision and recall are calculated. Obviously, for the first elements in the list, the recall will be low (it is calculated over all ground truths) while the precision could be near 100% (it is based on the detections, so if the first is a TP the precision will be high). The set of elements used to calculate those two values grows at each iteration, normally the precision will go down while the recall will go up as new elements are used. After all elements in the list have been used for this process, a point for each element is plotted in the graph. This produces the precision-recall graph. The area under this graph is the AP for the given class.
There are two methods to calculate the AUC (Area Under the Curve) for the AP: the *11-points interpolation* and the *all points interpolation* method.

11-points interpolation

The *11-points interpolation* gets the AUC value by averaging the graph values at eleven evenly spaced points.

$$AP = \frac{1}{11} \sum_{r \in \{0, 0.1, \ldots, 1\}} \rho_{interp}(r)$$

$$\rho_{interp}(r) = \max_{\tilde{r}:\tilde{r} \geq r} \rho(\tilde{r})$$

Where $\rho(\tilde{r})$ is the precision calculated at recall value $\tilde{r}$.

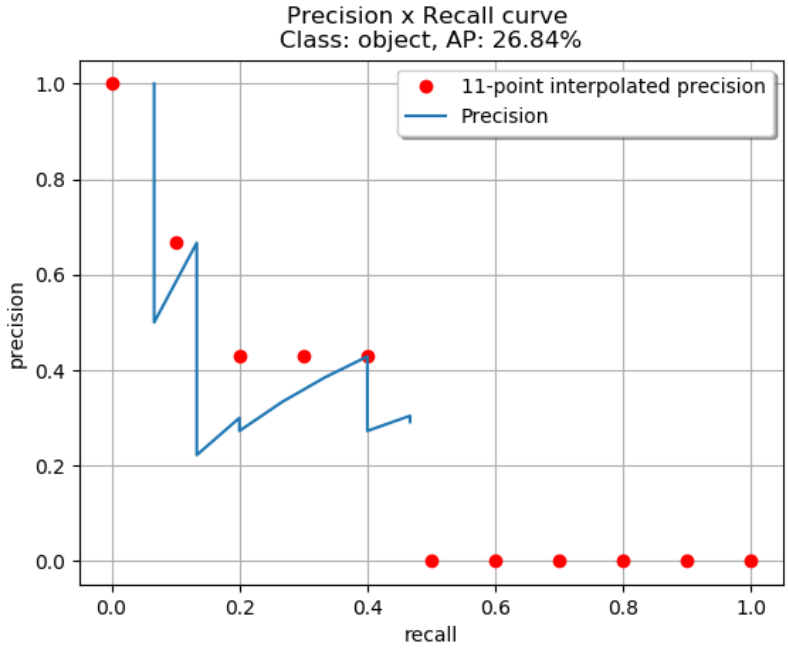


Figure 2.12: 11-points interpolation example [23]

As we can see from the figure 2.12, from the r value considered, the value of $p_{interp}(r)$ is obtained retrieving the maximum value of precision calculated on $\tilde{r}$

greater than r. This process to calculate the AP is then repeated for each class and averaged in order to get the mAP.

All points interpolation

Rather than interpolating only a few set of points, *all points interpolation* uses all points needed.

$$AP = \sum_{r=0}^{1} (r_{n+1} - r_n) \rho_{interp}(r_{n+1})$$
$$\rho_{interp}(r_{n+1}) = \max_{\tilde{r}:\tilde{r} \geq r_{n+1}} \rho(\tilde{r})$$

The following figure 2.13 shows the curve produced by selecting all points interpolation.

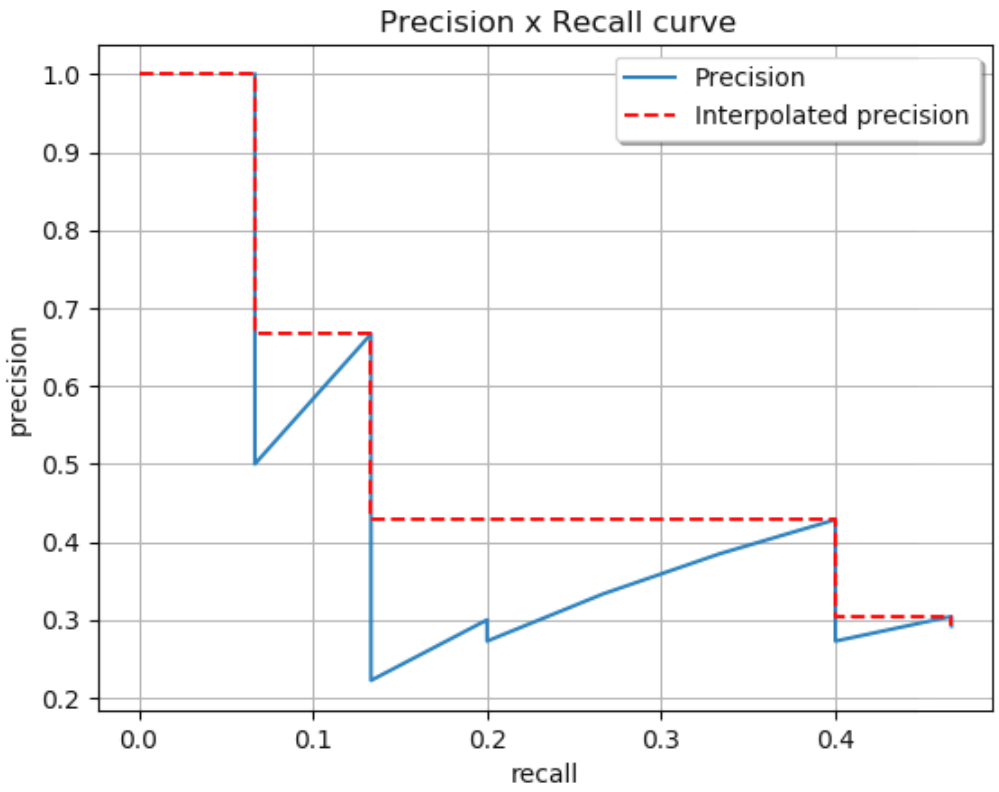


Figure 2.13: All points interpolation. [23]

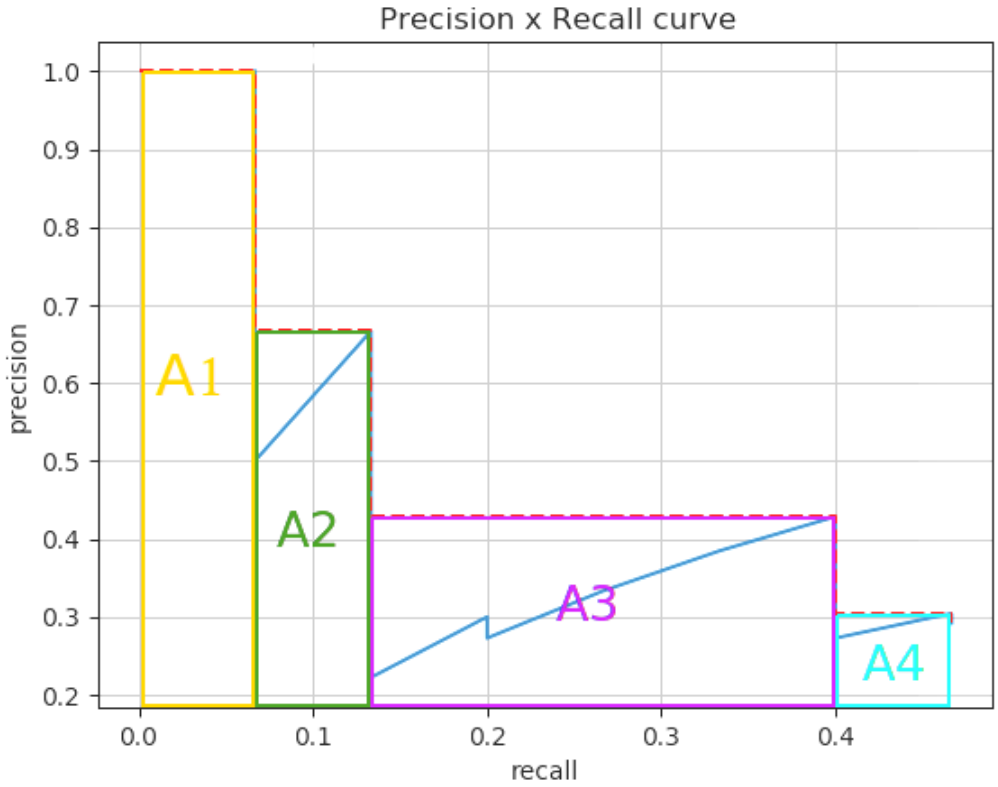


Figure 2.14: AUC results for all points interpolation [23]

As stated before, calculating the AP is the same as calculating the AUC for the *precision-recall* graph, we can see the process in figure 2.14. Then the AP calculation is repeated for all classes and then averaged.

2.4 Quantization

Deep neural networks are used in different tasks nowadays like computer vision, speech recognition, and robotics, with state-of-the-art accuracy[29]. Those results come at the cost of high computational complexity and energy consumption. Therefore, it is important to optimize these processes in order to be able to apply them on embedded systems.
Embedded devices come with limited computational capacity, limited storage and battery life (in case of smartphones). To try to solve these problems, in literature, there are different approaches both at the hardware level and at DNN algorithm level [19, 11, 20, 8, 24].
Quantization refers to the process of reducing the number of bits needed to represent a number on a computer, by mapping to a lower precision representation limiting the loss of information.

2.4.1 Values representation

In computer science, numbers are represented as sequences of bits. Real numbers are the most commonly represented values in machine learning and there are different methods to represent them:

- Floating point
- Fixed point
- Dynamic fixed point

A *floating point number* is composed of three parts: a sign bit, an exponent and a mantissa. The exponent represents the order of magnitude and the mantissa, the correct value in that order. For this reason, floating-point numbers can represent a wide range of values. There exist different floating-point formats :

Table 2.1: Floating point formats [4]

Format	**Total bit-width**	**Exponent bit-width**	**Mantissa bit-width**
Double precision floating point	64	11	52
Single precision floating point	32	8	23
Half precision floating point	16	5	10

Just by using the half precision floating point representation it is possible to reduce the bit width without having an impact on the training performance [4]. However, floating point operations are time and energy consuming from a hardware point of view.
Fixed point numbers are stored like integers in two's complement. Besides all fixed point numbers have a common scaling value, a shared exponent. This greatly simplifies the time and energy required to perform operations. Some embedded systems don't come with an FPU (floating point unit) so quantization can help also in this regard.
The last representation I consider is the *dynamic fixed point* that comes with a variable shared exponent. In Deep learning, it is used to have groups of variables with different scaling factors. For example layer's weights, bias, gradients, etc. can be used with a different scaling factor [4].

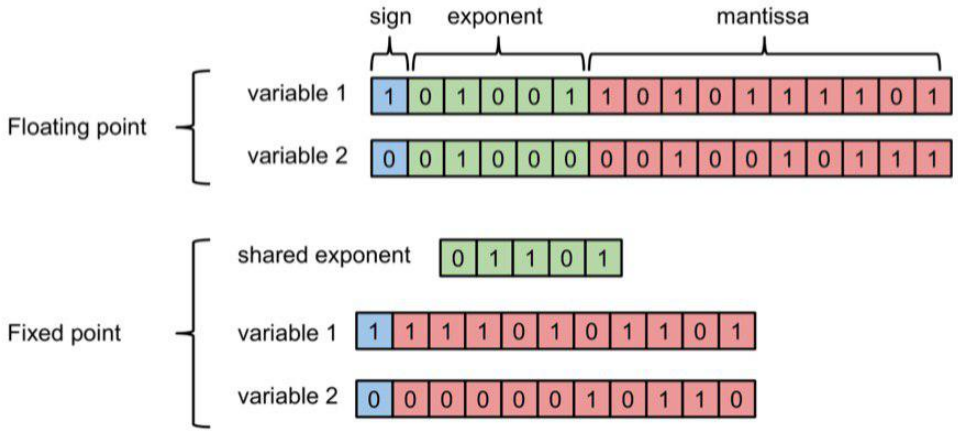


Figure 2.15: Floating point representation [4].

2.4.2 Uniform and Nonuniform

Quantization can be of two different types: *uniform* and *nonuniform*. The former is the easiest one and basically is a conversion from floating point to fixed point or to dynamic fixed point, this type preserves an equal spacing between consecutive represented value. The latter is not limited to equal spacing and it considers techinques like *log domain quantization* and *learned quantization or weight sharing* [30].

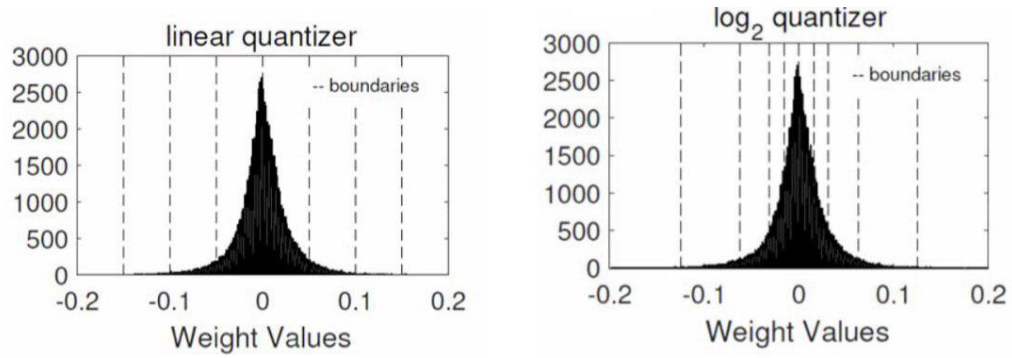


Figure 2.16: Uniform vs. NonUniform quantization [30]

This thesis focuses on uniform quantization, considering the dynamic fixed-point conversion. It is a linear mapping among a floating-point variable to a fixed point one. A limited range for the floating-point is linearly mapped, with some losses, to a fixed point representation. It is possible to apply the quantization both for weights and activations.
Quantization can improve a model performance both in terms of computational requirements and in model size. A low bit precision operation, clearly requires less energy to be performed increasing the battery life for mobile devices. It is also possible to easily build, custom hardware to improve efficiency. With some extreme techniques, it is possible to use one-bit weights, hence converting the convolutional

operation to a binary logic operation [24]. For what concerns the model dimensions, it is a common practice in machine learning to package a trained model as a single block ready to perform inference. Its size could potentially be a problem, particularly for embedded devices (for example Alexnet size is about 200Mb). With quantization, the size can be drastically reduced.

2.5 Tensorflow

In this section, I will describe briefly the important concepts of Tensorflow useful to explain this thesis work. Tensorflow is an interface for expressing machine learning algorithms and an implementation for executing such algorithms [1]. It has been released in 2015 by Google. As a high-level description, Tensorflow expresses computations as directed graphs. Nodes in these graphs are represented by operations and edges are data flows expressed as *tensors*, arbitrary sized arrays. One of the strengths of Tensorflow is the ability to run on a wide variety of heterogeneous systems (mobile devices, single machines or distributed systems) with little or no change to the algorithm specification[1]. It has also a built-in model training support.

2.5.1 Components

Variables are buffers that contain tensors, but are preserved between one execution of the graph and the other. Some variables properties are:

- They must be initialized.
- Tensorflow can use variables to perform training with gradient descent without additional code.
- Given that they are preserved between different runs, their values can be saved to disk.

For a single variable declaration, Tensorflow add three different operations:

1. An operation for producing the tensor, It corresponds to the initialization phase.
2. An *Assign* operation, it assigns the initialization tensor to the variable.
3. The effective variable operation, that holds the current value.

A **placeholder** is like a variable but without the need to initialize it immediately. Usually placeholders are used as input nodes.
Constants are nodes that take value that can't be changed.
An **operation** represents an abstract computation. A *kernel* in Tensorflow is a particular implementation of an operation that can run on a particular device.

Category	Examples
Element-wise mathematical operations	Add, Sub, Mul, Div, Exp, Log, Greater, Less, Equal, ...
Array operations	Concat, Slice, Split, Constant, Rank, Shape, Shuffle, ...
Matrix operations	MatMul, MatrixInverse, MatrixDeterminant, ...
Stateful operations	Variable, Assign, AssignAdd, ...
Neural network building blocks	SoftMax, Sigmoid, ReLU, Convolution2D, MaxPool, ...
Checkpointing operations	Save, Restore
Queue and synchronization operations	Enqueue, Dequeue, MutexAcquire, MutexRelease, ...
Control flow operations	Merge, Switch, Enter, Leave, NextIteration

Figure 2.17: List of common Tensorflow operations [3].

To interact with the graph, a Tensorflow program uses a **session**. It builds the graph, initializes all variables and is able to run a subsection of the graph to get an intermediate result, this type of execution is called *partial execution*.

2.6 Freezing a graph

In Tensorflow it is possible to save the state of a graph in order to persist its weights, hence the results of training. In this way, other people that want to use the graph, don't have to perform the training again. There are two possible ways. The first consist of a set of 4 files:

- A *.meta* file which holds the graph structure and the metadata.
- An *.index* file.
- A *.data* file which holds the weights.
- A checkpoint file.

With this format, in order to perform inference, it is necessary to have the source code.
The second way is to generate to a *.protobuff* file, a single file that holds all information needed by Tensorflow to perform inference. This format removes all unnecessary components, such as those related to the training process. For example, all Variable nodes are converted to Constant nodes.

Chapter 3

Related Works

Deep neural networks are widely used in various areas of Artificial Intelligence and their popularity is constantly increasing. This development has been paired with the increase in the computational power of CPU and GPU.
Due to their excellent results, nowadays the main concern is to find techniques to reduce the complexity of those algorithms to bring efficient deep neural networks to embedded devices without a constant connection to the internet.
To tackle this problem two different types of approaches have been used: hardware-based optimization and a software algorithmic optimization.
The first approach tries to develop a dedicated hardware structure to optimize data flows or improve parallelization. Parallelization of MAC operations (multipliers and accumulator) is the most commonly addressed, due to the large number of such operations involved in the inference phase of a deep convolutional neural network[30].
Moons et al. [17] describe some ad-hoc hardware solutions to reduce energy consumption and improve parallelization.
Andri et al.[2] described a hardware architecture for an accelerator optimized for binary-weights CNN.
Rastegari et al. [24] proposed a network binarization in two ways, a **Binary-Weight-Network** that affects only weights with a reduction in size with respect to an equivalent CNN of about 32 times, and a **XNOR-Network** with both weights and inputs binarized. XNOR-Network produces a 58x speedup in CPUs due to the reduced number of high precision operations. The advantage of binarization is that it is possible to execute convolution just by using binary logic operations.
Always related to weights binarization, **Courbariaux et al.**[5] proposed a method to train BNN (Binarized Neural Network) with weights and activation constrained to -1 and 1. This allows to drastically reduce memory size and to replace most arithmetic operations with bit-wise operation, reducing power consumption.
Moons et al.[19] developed a combined algorithmic and hardware system to improve energy consumption in common convolutional neural networks. The main idea is that common architecture widely use the ReLU activation function that produces 0 if the input value is lower or equal to 0. It is possible, by using hardware accelerators with dedicated hardware support, to skip ReLU computation in case of zero inputs. Besides, a precision scaling technique is implemented in hardware. For the algorithmic optimization, a *per-layer quantization* is used. The right quantization

value is obtained with a greedy search over the parameters, trying various quantization until the target accuracy is reached.
Gysel et al. [8] create Ristretto, a framework in Caffe, that is able to approximate floating-point operations using dynamic fixed point. This reduces the network size and energy consumption. Ristretto takes a trained model and performs a weight analysis, in order to calculate the dynamic ranges for dynamic fixed-point numbers. Then, performing inference with different images in input analyzes activation parameters for quantization. Afterward, the bit-width reduction phase produces an optimized model.
The reduction in network complexity can be achieved statically or dynamically, it means that at runtime it is possible to modify certain parameters to adapt the model to the processed data. **Jahier Pagliari et al.**[11] propose this approach applied on RNN (Recurrent neural network), where the parameter tuned at runtime is called BW (Beam Width). By increasing this parameter the network accuracy increases alongside the computational complexity. At runtime, the network is monitored and when necessary the BM is adjusted. They reduced the average BW by up to 33% with respect to a static network, while producing comparable or even better results.
A dynamic approach has been used also by **Tann et al.**[31]. Using hardware accelerators and a low-power embedded GPGPU, their technique is able to adjust the number of channels in the network. They achieved up to 95% energy reduction with less than 1% accuracy loss.
Regarding energy consumption optimization **Park et al.**[21] proposed an interesting combination of two different neural networks, one small and consuming low energy and the other a complete and high performant neural network. The idea is that processing input images don't have always the same complexity, so the first inference is performed using the small network. If the results are good enough (the network has good confidence in its prediction) the inference is complete. On the other hand, if the confidence score is too low, the inference is performed again with the bigger DNN. This approach has been called **Big/Little**. **Jahier Pagliari et al.**[10] propose another dynamic approach that exploits a bit-width reconfiguration, for CNN in object classification, via quantization. Their work needs a first phase in which the training set is analyzed in order to determine the values ranges to perform quantization. For each bit-width configuration, the inference is performed over the whole dataset. For each image, the least bit-width configuration that produced a confidence score greater than a certain threshold is recorded. During execution, the inference starts with the lowest bit-width quantization. If the result has a confidence score high enough, the execution moves to the next image. Otherwise, the bit-width is increased to repeat the inference. This process continues until either the confidence threshold has been exceeded or the maximum bit-width configuration has been used. In their work, they used two bit-width configurations at runtime.
This thesis work consists of two parts. The first concerns the creation of a software program using Tensorflow, able to, similarly to Ristretto [8], analyze a pre-trained model and add some fake quantization nodes with a given bit-width.
The second part is a proposal for an optimization for the Object detection network YOLOv2 [25], taking inspiration from the **Jahier Pagliari et al.**[10] work but applied for object detections. The final objective was to reduce the bit width, hence

the precision, of floating-point operations to optimize this highly efficient network to be used on an embedded device without a constant connection to the internet. First, a very low bit precision inference is performed, if some bounding boxes are predicted with a low confidence score, the inference is performed again with a higher bit-width and over a region of the image that contains the predicted bounding box.

Chapter 4

Arbitrary Bit-Width Post Training Quantization in Tensorflow

To adapt Deep Neural Networks to embedded systems, limited in hardware performances, there are two main approaches: the creation of custom hardware and circuits or logic optimizations at the algorithm level. For the second case, as explained in chapter 2, there are different techniques. They can be further divided into two categories: static or dynamic approaches. The former tunes some network parameters that are kept constant during inference while the latter update them dynamically as the input changes.
The final objective of my thesis is to develop a program in Tensorflow which is able to apply this dynamic approach to an object detection network, YOLOV2[25]. Specifically, an approach similar to the **Jahier Pagliari et al.**[10] work, with a first inference performed on a low bit-width quantized network and a refinement performed, if necessary, with a full-fledged network.
In this chapter, I explain the first phase of the work: the analysis of a pre-trained model and the quantization of network operations. The objective is to simulate the effect of running a quantized inference on an embedded device, using a normal PC with TensorFlow. To this end, a “fake quantization” is performed. The "fake quantization" provides us the information that we need, it is taking into account, in the network calculations, the precision loss that would be present using variables stored with a reduced bit-width, even if they are stored in memory as full-fledged 32 bits float. Importantly, the fake quantization implemented can support arbitrary bit-widths (for example, not only 8 and 16 bit, which are also supported natively by Tensorflow, but also intermediate bit-width such as 9, 10, 11, etc.).
I tested the quantization on a classification task using AlexNet [12] and the ImageNet validation set [6] for the LSVRC-2012 competition. Results will be measured using top-1 and top-5 accuracy.

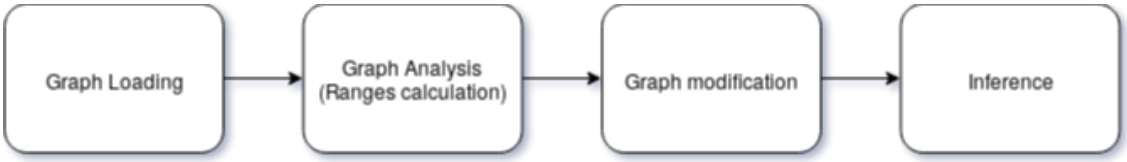


Figure 4.1: The program flowchart.

4.1 Graph Loading

As I described in chapter 2, Tensorflow stores network models in two ways: a set of .index, .meta, .data and checkpoint files or a Protocol Buffer file .pb containing the description of a frozen model. Protocol buffers are Google's mechanism for serializing structured data in a way that is language and platform-independent. It is a similar concept to XML. This format is used in Tensorflow to store a frozen graph. A .pb file contains all that is necessary to use the pre-trained model for inference. Given that it is basically a text file, firstly a parsing phase is needed to restore the graph by populating a Tensorflow data structure called *graph def* (graph definition). The program begins by loading the .pb file and then it recovers the graph definition, in this way it will be possible to apply the various changes needed. A graph in Tensorflow is composed of operations that represent transformations applied to tensors. Once the graph is restored, it is possible to retrieve all operations and analyze them. Each operation comes with various attributes, the most important:

- Name: by default is $<op\_type>\_<x>$, where x is the number of occurrences of the same operation type in the graph.
- Type: the type of operation as described by 2.17.
- Inputs and Outputs: lists of tensors in input and in output, they represent the edges that connect the nodes.

In order to have easily access to the graph, I created a class named *FrozenGraphHandler* that simply loads the definition and extracts references to the input and output tensors, needed to perform inference in the next stages.

```
1     # Read the graph definition
2     with tf.io.gfile.GFile(self.frozen_model_path, "rb") as f:
3         graph_def = tf.compat.v1.GraphDef()
4         graph_def.ParseFromString(f.read())
5
6     # Restore the graph
7     with tf.Graph().as_default() as graph:
8         tf.import_graph_def(graph_def,
9                             input_map=None,
10                            return_elements=None,
11                            name=""
12                            )
13        self.graph = graph
```

```
14
15      # Save first and the last operation,
16      # the order may change after the modifications
17      self.input_tensor = self.graph.get_operations()[0].outputs[0]
18      self.output_tensor = self.graph.get_operations()[-1].outputs[0]
```

In this listing, we can see the code used to read the .pb file, restore the data and store the input and output tensors. The first operation receiving the input tensor while the last operation returning the output one.

4.2 Graph Analysis

The next phase is the graph analysis, performed to determine, for all selected nodes, the ranges of possible values assumed by the activations for the current dataset. This is used in the next phase to apply quantization.
The **FrozenGraphHandler** has a reference to the restored graph that will be use to have access to all operations. By looking at 2.17, it is possible to see that not all nodes need quantization, for example, Less, Greater, Shape, etc. The quantized nodes will be Add, Mul, Matmul, and Conv2D. For this thesis work, they were sufficient.
The class that performs the analysis is called **FrozenGraphAnalyzer**. Initially, it iterates over all graph operations and it saves a reference to all nodes having an operation type present in the list mentioned above.
Then the range calculation is performed exploiting the partial execution functionality in the Tensorflow session. By passing a reference to all selected tensors and the input images, the *session.run()* methods return a list of all partial outputs, one for each requested operation and each input image. Then, after having determined the min and max value over all inputs for each operation, it builds a dictionary that will be used in the graph modification phase. The dictionary contains an entry for each node, where the key is the node name and the value is:

- *min_value*
- *max_value*
- *num_channel*: The number of channels in the node output. The quantization node must keep the same number.
- *n_bits*: The bit-precision.
- *output_tensor*: Reference to the output tensor.
- *op*: Reference to the operation.

The *n_bits* is present because a possible improvement in the program would be a *per-layer quantization*. It means to change the bit precision, based on the range values for different layers.

4.3 Graph Modification

The graph modification phase is handled by the class **FrozenGraphModifier** that needs a reference to the **FrozenGraphHandler**, to access the graph, and to the dictionary built in the previous phase. The modifier creates a new graph node for each operation to quantize and then it inserts it into the graph.

4.3.1 Fake Quantization

In this work, I use fake quantization. It means that the node class, that I will describe shortly, is taking the input represented in floating-point, it is clamping its value, based on the selected bit precision, then it is converting it back to floating-point before producing the output.
The class that defines the Tensorflow custom node is called **BaseRoundingQSim** and its base class is **QSim** (from Quantization Simulation). The hierarchical structure is necessary to support, in future works, other types of quantization (logarithmic, etc.).
BaseRoundingQSim needs all data contained in the dictionary for the operation that it has to quantize. In Tensorflow it isn't possible to use standard python code in the graph unless by using a particular annotation: **@tf.function**. This tells Tensorflow that the following method has to be used as a graph node, it will be wrapped and inserted in the model definition. The method marked with **@tf.function** is the class-default *__call__()* method. It is suggested to use this structure to avoid problems with function scope in case of variable declaration inside the method.
The class uses a function provided by Tensorflow to perform the mapping:
tf.quantization.fake_quant_with_min_max_vars_per_channel()
This function takes the starting range as input $[a, b]$, and the number of bits to define the mapping. Given n bits, there are only 2^n possible value to be represented. The TensorFlow function maps from $[a, b]$, to $[0, 2^n - 1]$. Then it converts back to $[a, b]$ range, with a loss in precision that simulates the real quantization at a low bit representation.

4.3.2 Graph Modification

After having created a quantization node it is time to insert it in the graph. As I described in chapter 2, the Tensorflow graph is composed of nodes connected by edges. Each edge is a Tensor, data that flows from one node to the other. Adding a node means that some edges have to be rerouted, disconnecting all inputs in the following node and connect them to the new node, then connect the latter's output to the former's input. This is a complex procedure, also because Tensorflow isn't meant to be used in this way. The structure of a general Tensorflow program is divided in a first phase of graph definition followed by the execution via a *session.run()* call. This suggests that after a graph is composed, the modification should be made only by modifying the code structure in the composition phase, not in the execution phase. However, our goal is to be able to modify pre-trained models for which only the frozen graph is available. Therefore, there is no alternative but to modify the

pre-generated graph. Fortunately, in the *tf.contrib* module (which contains various packages built by the community, being Tensorflow an open-source project), there is a package called **graph_editor** that enables the kind of modifications that are needed to add and remove nodes.
Graph_editor uses **SubGraphViews**. As the name suggests they are a useful tool to monitor the state of a group of nodes in the graph. It is possible to have a SubGraphView of only one node. After the creation of that structure, it is possible to use various methods to insert and modify other SubGraphViews. The *Graph_editor* can't act directly at the node level.

```
    # Get the subgraph view, it is needed by Graph_Editor
    #   to perform modifications
    op_to_add_sgv = ge.sgv(op_to_add)
    prev_op_sgv = ge.sgv(data_for_op[Constants.DICT_KEY_OP])

    # Connect op to the new operation exchanging
    #   the output
    ge.swap_outputs(prev_op_sgv, op_to_add_sgv)
    ge.connect(prev_op_sgv, op_to_add_sgv)
```

Ge is an object of type *graph_editor*. *Data_for_op* is the dictionary containing all useful data for the quantization. The code performs the node insertion. *Op_to_add_sgv* contains the new node to be added while *prev_op_sgv* contains a reference to the node to be modified. The *swap_outputs()* method is swapping the outputs of the previous node and the quantization node. Given that the quantization node is isolated, *swap_output()* is detaching the previous node output from the graph. Then *connect()* is used to reconnect the graph.

4.4 Alexnet Analysis

In order to test the program results and the effect of a quantization on a real case, I used the AlexNet CNN [12] on the ImageNet validation set [6]. In order to evaluate the performance, I used the top-1 and top-5 accuracy. Accuracy is calculated as follows:

$$Acc = \frac{\text{Correct predictions}}{\text{All predictions}}$$

Alexnet's output is an array of probabilities, one for each class. In the case of Imagenet, there are 1000 classes. The difference among top 1 and top 5 is the fact that top 1 considers only the prediction with the highest probability whereas top 5 considers the best five.
The inference result can be seen in the picture below:

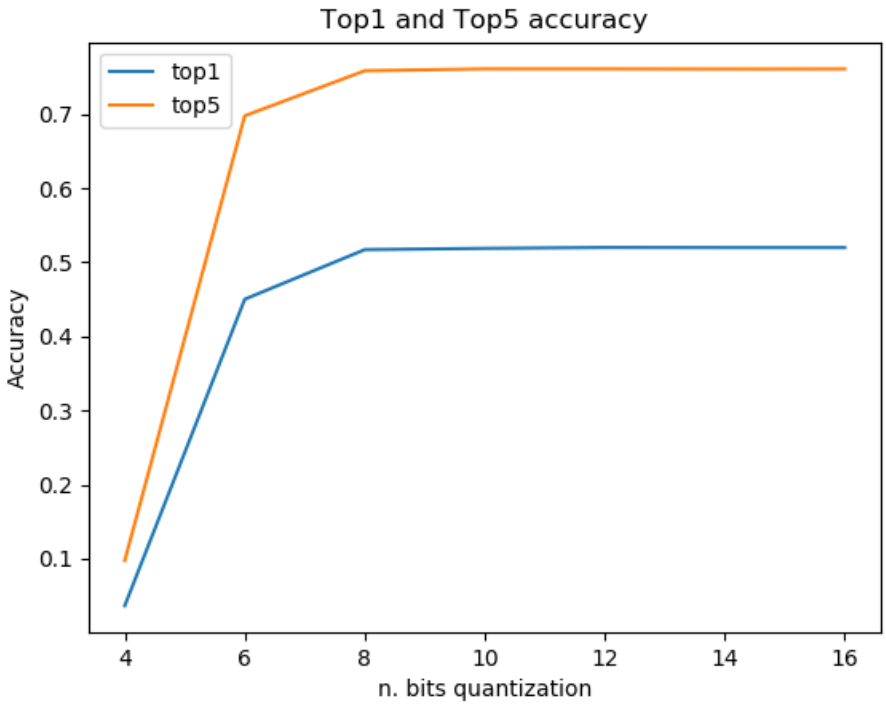


Figure 4.2: Alexnet[12] top 1 and top 5 accuracy on ImageNet[6].

It is possible to notice that going below 8 bits, the loss in accuracy is very high.

Chapter 5

Selective Refinement for Energy-Efficient Object Detection

In this chapter, I will describe the second part of this thesis work. Based on the software described in the previous chapter, the final goal is to build a system to enhance CNN performances on embedded devices mainly in terms of energy consumption. The program uses a similar approach to the **Jahier Pagliari et al.**[10] work but applied on an object detection task. I use a first CNN having half of all layers quantized at 4 bits and the other half at 8 bits. It is used to perform inference over an image. By itself, the accuracy of this network won't be very high. For that reason, in case of a box confidence score lower than a certain threshold, a second network is used with higher precision. This high precision network, however, will only be executed on parts of the image containing the poorly defined boxes. Therefore, on average, this network will perform fewer operations with respect to a solution that uses it directly on the original image. As a consequence, the energy consumption and latency of the overall inference will be reduced.

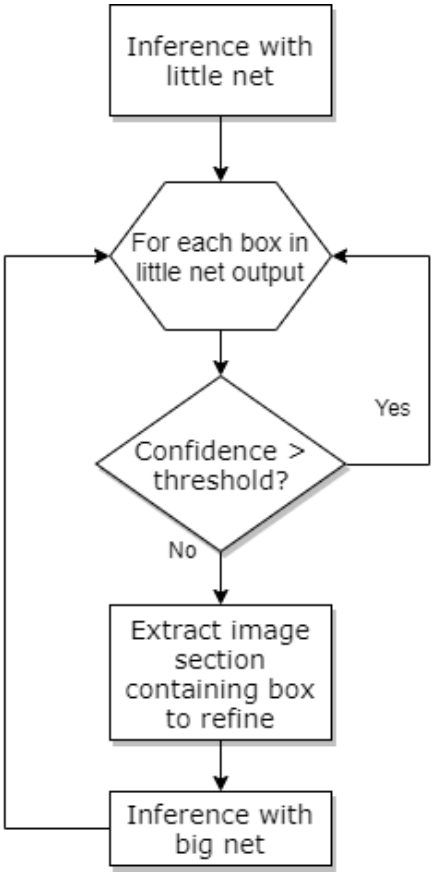


Figure 5.1: Sequence Diagram.

5.1 Procedure

The CNN used for this thesis, as explained previously, is YOLOv2[25] due to the good performances in object detection and its structural simplicity. The low precision network has the first half of its layers quantized at 4 bits, the other at 8 bits. This network is built and used to perform inference on the complete image $416 * 416 * 3$. The output will contain some bounding boxes definition, together with the box confidence score which is expressed as:

$$\text{Box confidence score} = \sigma(t_0) = Pr(object)$$

This represents a measure of how good a bounding box is. This inference will produce a set of bounding boxes that won't have high values of confidence. If that value isn't enough, the second network will be used to "refine" the prediction. The latter is an 8-bit quantized YOLOv2 network that can have very accurate predictions. The input to this network will be a section of the initial image. Figure 5.2 shows the process execution.

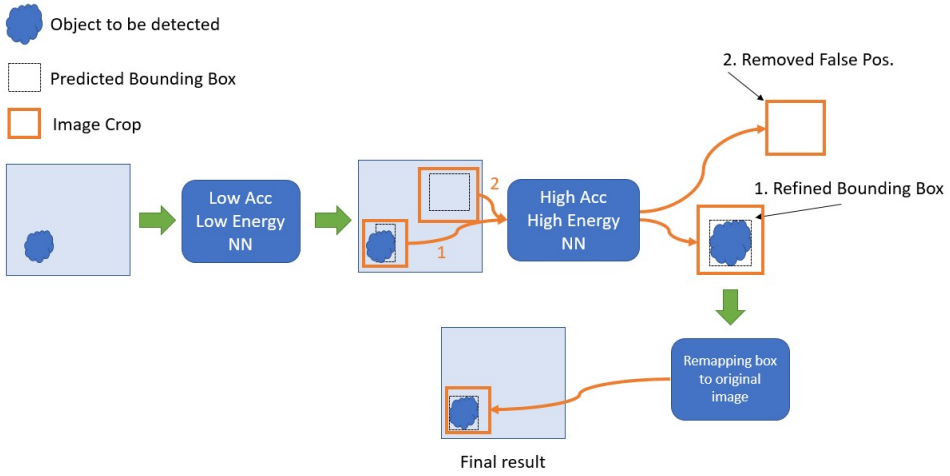


Figure 5.2: The program flow graph.

This comes with some problems. First of all, we have to consider that the main pattern of the YOLOv2 structure is: Conv2D-Relu-Pooling and it is repeated multiple times with some cases with more than a single Conv2D layer. I can't use an arbitrary size in input, but I have to choose a dimension that allows all layers to execute their tasks without problems.
Let's start analyzing the Conv2D layers to understand what sizes are accepted. Each one has a padding of type *'same'*, which means that zero padding is added to maintain the same size among input and output, and is paired with a kernel. Filters come with two different dimensions: a $3*3*x$ (where x is the input depth) and the other $1*1*x$. Considering how the convolution is executed, a filter $1*1*x$ can be applied no matter the input size. The same is true also for the $3*3*x$ filter considering that the padding mode is *'same'*.
Relu layer doesn't pose any problem. For the MaxPooling layer, we have to be careful about the padding. If the input dimension is odd, with a stride of 2, the rightmost columns are discarded without proper padding. Usually, the padding is applied in order to avoid to discard some values. Anyway, it is possible to calculate the accepted inputs and outputs dimensions without problems. Finally, the YOLOV2 structure contains a *passthrough* layer that turns a $26*26*x$ layer into a $13*13*4x$ layer by dividing the original by 2 over the x and y axis. This block is then concatenated with the layer just preceding the *passthrough*. Considering the division, it is clear that width and height for the input tensor must be even, else the concatenation will fail (sizes won't be compatible). Knowing all that, it is easy to calculate all possible values accepted in input.
As explained in chapter 2, the YOLOV2 output expresses a bounding box using 4 values: a center (x, y), width and height. Those values are expressed as offsets with respect to the top-left corner of the grid cell, for the center point, and to the most similar anchor box for width and height. In case of a confidence score lower than a certain threshold, it is possible to refine the box by extracting a section of the image containing the previously predicted box. The section sizes have to be among

the acceptable values, else the YOLO prediction will fail. If the box doesn't match any acceptable value, a bigger section is extracted maintaining the center of the box as the center of the section. This part of the original image is used as input for the second network. The result is analyzed and different situations could happen:

- The network didn't find any box. In this case, also the first network prediction is discarded.

- The network found a box, it must be remapped to the original image.

- The network found more than one box. All boxes are added to the prediction list and they are remapped to the original image.

After the refinement, the YOLO output has to be remapped to the original image. This is straightforward because, from the extraction phase, I still have access to the top-left corner of the section of the image. Firstly I have to express the box prediction in terms of pixel coordinates within the sub-image. Then by simply summing those values with the top-left corner of the extracted image, expressed in pixel coordinates, we will remap the bounding box values in the original size image.

It is clear that this thesis work is trying to exploit two different optimizations:

- The reduction in energy consumption due to the reduction in bit width for convolutional operations.

- The reduction in size, for convolutional layers, due to the reduced input size derived from the extraction phase.

The second optimization isn't a great deal for the majority of general object detection tasks, in fact, it is common to have boxes that occupy a great portion of the original image. In those cases, the extracted section won't provide almost any advantages, the size reduction would be negligible. This technique has a great potential though if the dataset contains small objects that don't fit the whole image. The application case for this thesis work considers a dataset of images similar to figure 5.3.

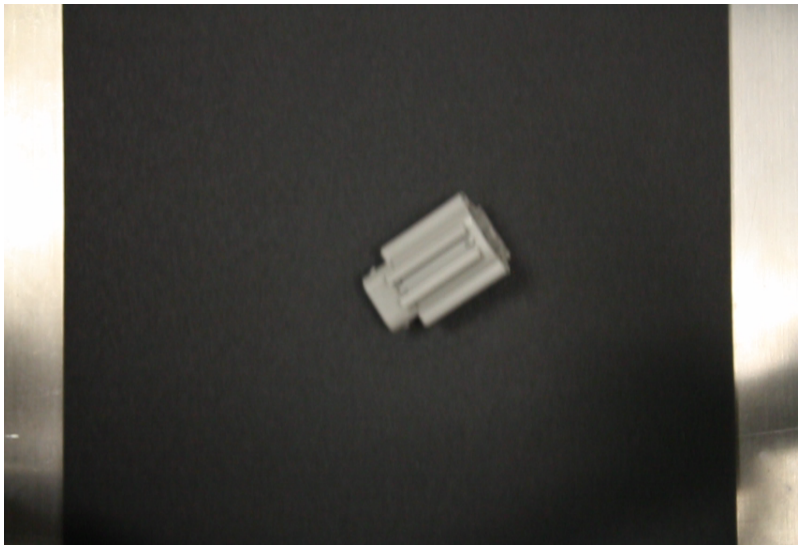


Figure 5.3: Use case dataset example.

The object occupies a very small part of the image, so the extraction phase will greatly reduce the new input size. These images contain industrial components (plastic connectors) on a conveyor belt. The objective is to detect them in order to perform counting and automatic quality inspection.

5.2 Evaluation Procedure

5.2.1 Datasets

In order to evaluate the performance of this method, I used a realistic industrial dataset, generated in the context of a European project, which contains images of objects moving on a conveyor belt, which have to be detected in order to perform automatic counting. In the images, objects are small and few per frame. Three different datasets have been used:

1. Set of images with at most one single object.
2. Set of images with at most two objects.
3. The complete validation set with one, two and more objects.

The best results are expected for Dataset 1. This depends on the fact that having a single object, the extracted image will be a minor portion of the image. For the other two cases, for the presence of multiple objects, there will be multiple refinements with the consequent increase in energy consumption.

5.2.2 Metric

Generally, all object detection tasks, as explained in chapter 2, are evaluated using the mAP score. That score isn't well suited to evaluate this particular task. Mean

average precision counts all boxes greater than a certain threshold as equally valid, but given that my work is refining a box, it would be better to find a metric that keeps into account also the box quality. To perform an accurate analysis of the proposed solution, a set of different metrics has been used.

- TP: True positive.
- FP: False positive.
- FN: False negative.
- IoU: Intersection over the union, averaged over only TP.
- Energy spent to complete the task. It is an average over all images in the dataset.
- mAP: mean average precision.

The IoU is a crucial value, it gives information on how good is a bounding box and by averaging it over only the correct predictions, I can get a good estimate of how good it is the refinement process.
TP measures how many boxes have an IoU greater than a threshold with respect to the ground truth.
FP measures how many boxes aren't greater than that threshold. Besides, boxes with IoU greater than the threshold for ground truth already matched. A predicted bounding box with IoU greater than the threshold has been already founded for this ground truth.
FN measures how many ground truth boxes don't have a corresponding prediction with IoU greater than the threshold.
In order to estimate the energy consumption, I used the work of **Moons et al.** [18] that provides a table with values of power consumption for MAC (multiply-accumulate operation), all other operations are negligible due to the higher number of MAC needed for convolutions. The formulas to get the energy for a single MAC at a given bit width are:

$$Emac\_8 = 0.103 * \frac{1}{250 \cdot 10^6} * \frac{1}{128}$$
$$Emac\_4 = 0.045 * \frac{1}{125 \cdot 10^6} * \frac{1}{256}$$

The leftmost factors (0.103 and 0.045) in the multiplications are the power values reported in [18] (expressed in Watts) while the rightmost factors are added because the hardware described in that work performs 128 8-bit multiplications or 256 4-bit multiplications in parallel. When doing the former, the hardware is able to run at 250MHz, while for the latter operation it runs at 125MHz. Thus, the middle factors in the two multiplications represent the time required for each operation and are used to convert the power values into energy values. In this way, it is possible to sum all contributions for each MAC operation either at 4 bits or 8 bits.

These quantities have to be multiplied by the number of MAC operations performed and for the number of images to get the total energy:

$$Tot\_energy = Emac * num\_op * num\_img$$

In order to calculate the number of operations I used the following formulas:

$$Tot\_op\_conv\_layer = N_i * N_j * S^2 * (W * H) * (W' * H')$$

Where:

- N_i: the number of feature maps in input. In other words, the depth of the input tensor.
- N_j: the number of feature maps in output.
- S: The filter size. For example, in a $3 * 3$ filter, this value will be 3.
- W and H are the width and height for the input tensor.
- W' and H' are the width and height for the output tensor.

This formula is applied for all convolutional layers. The number of operations depends on the input sizes. So by reducing the input dimensions, it is possible to reduce the number of operations hence the energy consumption.
Lastly, there are some threshold values to set up in order to use YOLO and to calculate the mAP:

- ThC: Confidence threshold. YOLO considers a box as a valid prediction, only if the box confidence score is greater than this value. It means that YOLO thinks that there is an object inside the box.
- ThS: Selective refinement threshold. Looking at figure 5.1, it is the threshold used to determine if refinement is needed or not.
- ThO: Overlap threshold. This value is used during non-max suppression. All boxes with an IoU greater than this value with respect to the box with the highest confidence score are discarded.
- ThP: True positive threshold. During the mAP calculation, the number of TP is calculated. This value is the threshold to consider a box as a TP or not.

5.2.3 Test Runs

In order to have a good understanding of how good is this optimization technique I performed the analysis with three different setups:

1. A complete 8-bits network. This will provide the highest accuracy in the test.
2. A YOLOV2 with the first half layers quantized at 4 bits and the other at 8. This will provide the lower bound for the test.
3. The refinement process showed in figure 5.1.

I will expect to see case 3 to have an accuracy greater than the second setup while having a similar energy consumption value.

5.2.4 Grid Search

In order to check what are the best values for those threshold, I performed a grid search with the following ranges:

$$ThC = [0.05, 0.6, 0.05]$$
$$ThS = [0.5, 1.05, 0.05]$$
$$ThP = [0.5, 1, 0.05]$$

Following python notation for *range()*, the first value is the range starting value, the second is the final value exclusive and the third is the step amount. It means that after $ThC = 0.05$ the following value will be $ThC = 0.1$. The ThP is used to calculate the mAP values that are averaged to get the final result.

5.2.5 Pareto Frontier

In order to evaluate if the refinement performed better than the other two networks, I based the analysis on mAP and energy consumption. Having a multivariable optimization problem, the Pareto front is used to find all those couples of values that are considered the best in the dataset. Specifically, for multiple variables, the best values are those couples for which it isn't possible to improve one component without penalizing the other. In this case, for a Pareto element in this dataset, it isn't possible to improve the energy component without penalizing the mAP. In this way, if the refinement network produces values that exceed this line it is clear that they are better results with respect to the other 2 network outputs. The side on which a refinement output is considered better depends on whether it is better to have a higher value, for a component, or not. The *Pareto front* is calculated over the 8 bits and half half networks.

Chapter 6

Results and Future Works

6.1 Results

In this section, I will show graphs Energy-mAP. The energy is calculated in Joule and it is the average over all images in the dataset. The mAP is the average of all mAP calculated at different IoU with threshold *ThP*. The graphs will show the best result obtained using a certain combination of *ThC* and *ThS* (clearly *ThS* only for the refinement case).

6.1.1 One object dataset

This dataset contains images with at most a single object. There are about 40 images.

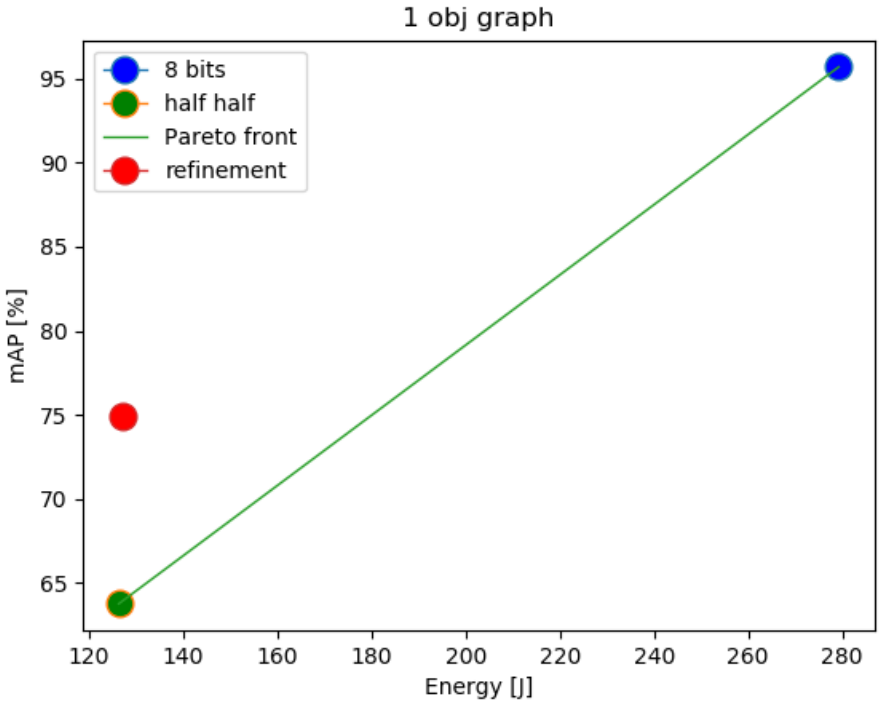


Figure 6.1: Results one object dataset.

From figure 6.1 it is possible to see that, with an energy consumption almost equal to the half half network, it is possible to achieve better mAP.
Following, the values for the figure 6.1:

Table 6.1: Figure 6.1 results.

	Energy [J]	**mAP[%]**
8 bits network	279.252	95.697%
refinement	127.336	74.957%
half half network	126.385	63.769%

6.1.2 Two objects dataset

Figure 6.2 shows results relative to the second dataset. It contains images with at most 2 objects (it contains also all images of the first dataset). There 313 images in this dataset. With respect to the previous graph, here we can notice that the overall mAP is reduced from about 75% to around 70%. However, this is true also for the original 8-bit and half/half networks, so the benefit of the proposed refinement method is similar to the previous case (around 10% improvement).
Following, the values for the figure 6.2:

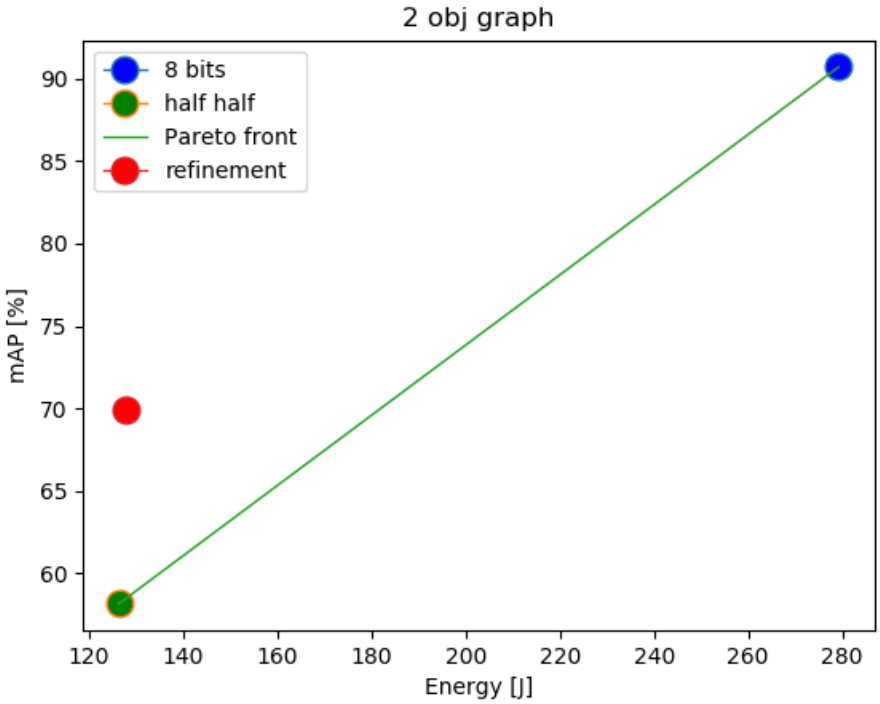


Figure 6.2: Results two objects dataset.

Table 6.2: Figure 6.2 results.

	Energy [J]	**mAP[%]**
8 bits network	279.252	90.708%
refinement	127.854	69.92%
half half network	126.385	58.154%

6.1.3 Validation dataset

Figure 6.3 shows results relative to the last dataset, the validation set. It contains all images of the first and the second dataset, in addition it contains images with more than two objects. There 961 images in this dataset. In this case the difference in terms of energy consumption is slightly more visible. This happens because there are images with more than two objects, so the refinement could be performed multiple times for a single image, one for box.
Following, the values for the figure 6.3:

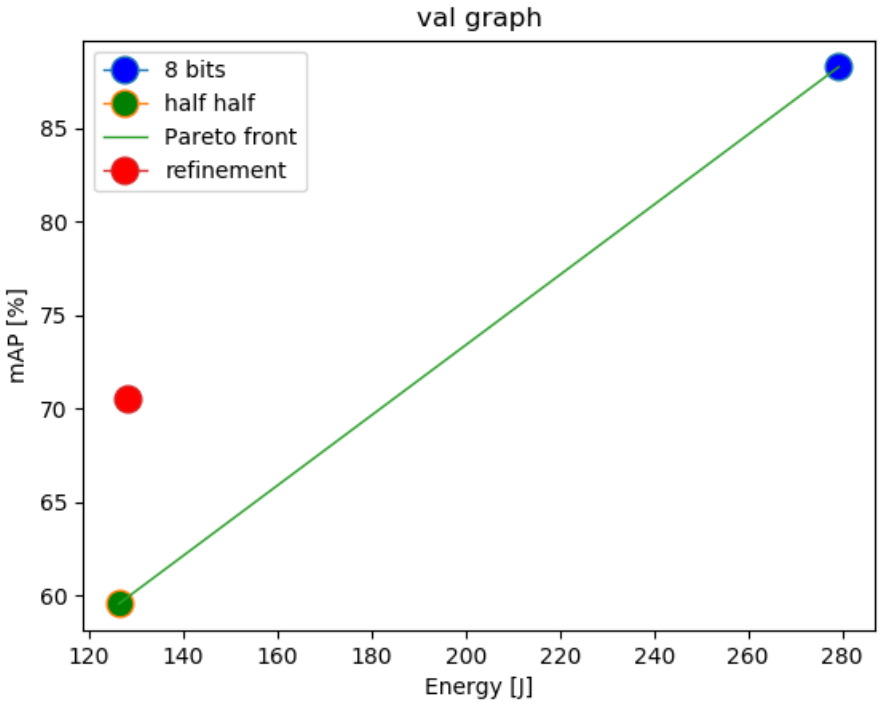


Figure 6.3: Results validation dataset.

Table 6.3: Figure 6.3 results.

	Energy [J]	**mAP[%]**
8 bits network	279.252	88.256%
refinement	128.247	70.553%
half half network	126.385	59.572%

6.2 Future Works

In this section I will explore some of the improvements that can be applied to this thesis work:

- Implement different quantization methods.
- Switch to Tensorflow 2.0.
- Combination of different image section for refinement.

6.2.1 Quantization methods

As explained in chapter 2 there are different quantization methods. In the context of this thesis work, it would be possible to apply one of the *nonuniform* techniques like *log domain quantization* or *learned quantization or weight sharing* [4]. As shown

in chapter 4 the **QSim** is an easily extendable class. It is sufficient to implement **@tf.function** for the **__call__()** method.

6.2.2 Tensorflow 2.0

Recently Tensorflow version 2 has been released. It brings lots of changes to the Tensorflow 1 workflow explained in chapter 2 and in order to move this project to that version there are different things to consider. In general, with this release, Google wanted to realize the following points [32]:

- Fewer lines of code.
- Increased clarity and simplicity.
- Easier debugging.

The concept of *eager execution* by default, has been added. It means that the call to *tf.session.run()* isn't necessary anymore, instead, we have to wrap our operations into python functions and run them as simple methods.
The usage of name-based variable tracking is discouraged in Tensorflow 2.0. I used it extensively to retrieve references to network nodes in my implementation. This could be a big obstacle to the conversion to Tensorflow 2.0. It is more difficult to access all information contained in the *proto buffer* file of the network. In addition the *.pb* format isn't the standard way to store a frozen model anymore. Instead a **SavedModel** format is used.
The *tf.Contrib* module has been removed in Tensorflow 2.0. Some packages have been converted, but **graph editor** did not. As shown in chapter 4 I used it extensively to easily modify the graph by adding nodes. This adds another obstacle to the conversion, in addition to the new difficulty to find a reference to a given layer, it is really hard to modify the graph. All modifications has to be made directly on the *proto buffer* graph description file.
The usage of the *@tf.function* annotation is still present.

```
1     W = tf.Variable(tf.ones(shape=(2,2)), name="W")
2     b = tf.Variable(tf.zeros(shape=(2)), name="b")
3
4     @tf.function
5     def forward(x):
6         return W * x + b
7
8     out_a = forward([1,0])
9     print(out_a)
```

The listing above shows a simple example that explains how the new programming workflow for Tensorflow 2.0 can be used.

6.2.3 Combined refinement of multiple image sections

This thesis results have been achieved using a particular dataset. All images contain small objects. This is useful because, in this way, the resulting bounding boxes will be small as well. Then, by extracting from the image the corresponding section with the accepted dimensions, I am sure that there will be a great reduction in the number of operations for the convolutional layers. In the case of multiple objects, I'm performing one refinement for each extracted section. One possible improvement could be an algorithm to join together nearby sections in a way to create a bigger image containing multiple boxes that still respects the constraints in size. In this way, the number of operations will be bigger, but the refinement will be performed just once, possibly eliminating some repeated computations and further improving efficiency.

Bibliography

[1] Martín Abadi et al. *TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems*. 2016. arXiv: 1603.04467 [cs.DC].

[2] R. Andri et al. "YodaNN: An Architecture for Ultralow Power Binary-Weight CNN Acceleration". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 37.1 (2018), pp. 48–60. DOI: 10.1109/TCAD.2017.2682138.

[3] Nikhil Buduma and Nicholas Locascio. *Fundamentals of deep learning: designing next-generation machine intelligence algorithms*. OóReilly, 2017.

[4] Matthieu Courbariaux, Yoshua Bengio, and Jean-Pierre David. *Training deep neural networks with low precision multiplications*. 2014. arXiv: 1412.7024 [cs.LG].

[5] Matthieu Courbariaux et al. *Binarized Neural Networks: Training Deep Neural Networks with Weights and Activations Constrained to +1 or -1*. 2016. arXiv: 1602.02830 [cs.LG].

[6] J. Deng et al. "ImageNet: A Large-Scale Hierarchical Image Database". In: *CVPR09*. 2009.

[7] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. http://www.deeplearningbook.org. MIT Press, 2016.

[8] Philipp Gysel, Mohammad Motamedi, and Soheil Ghiasi. *Hardware-oriented Approximation of Convolutional Neural Networks*. 2016. arXiv: 1604.03168 [cs.CV].

[9] Sergey Ioffe and Christian Szegedy. *Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift*. 2015. arXiv: 1502.03167 [cs.LG].

[10] Daniele Jahier Pagliari, Enrico Macii, and Massimo Poncino. "Dynamic Bit-width Reconfiguration for Energy-Efficient Deep Learning Hardware". In: July 2018, pp. 1–6. DOI: 10.1145/3218603.3218611.

[11] Daniele Jahier Pagliari et al. "Dynamic Beam Width Tuning for Energy-Efficient Recurrent Neural Networks". In: May 2019, pp. 69–74. DOI: 10.1145/3299874.3317974.

[12] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. "ImageNet Classification with Deep Convolutional Neural Networks". In: *Advances in Neural Information Processing Systems 25*. Ed. by F. Pereira et al. Curran Associates, Inc., 2012, pp. 1097–1105. URL: http://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf.

[13] Y. LeCun et al. "Backpropagation Applied to Handwritten Zip Code Recognition". In: *Neural Comput.* 1.4 (Dec. 1989), pp. 541–551. ISSN: 0899-7667. DOI: 10.1162/neco.1989.1.4.541. URL: http://dx.doi.org/10.1162/neco.1989.1.4.541.

[14] Y. Lecun et al. "Gradient-based learning applied to document recognition". In: *Proceedings of the IEEE* 86.11 (1998), pp. 2278–2324. DOI: 10.1109/5.726791.

[15] Yann Lecun, Yoshua Bengio, and Geoffrey Hinton. "Deep learning". In: *Nature* 521.7553 (2015), 436–444. DOI: 10.1038/nature14539.

[16] Andrew L. Maas. "Rectifier Nonlinearities Improve Neural Network Acoustic Models". In: 2013.

[17] B. Moons et al. "14.5 Envision: A 0.26-to-10TOPS/W subword-parallel dynamic-voltage-accuracy-frequency-scalable Convolutional Neural Network processor in 28nm FDSOI". In: *2017 IEEE International Solid-State Circuits Conference (ISSCC)*. 2017, pp. 246–247. DOI: 10.1109/ISSCC.2017.7870353.

[18] Bert Moons et al. "DVAFS: Trading computational accuracy for energy through dynamic-voltage-accuracy-frequency-scaling". In: *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2017* (2017), pp. 488–493.

[19] Bert Moons et al. "Energy-efficient ConvNets through approximate computing". In: *2016 IEEE Winter Conference on Applications of Computer Vision (WACV)* (2016). DOI: 10.1109/wacv.2016.7477614. URL: http://dx.doi.org/10.1109/WACV.2016.7477614.

[20] Bert Moons et al. *Minimum Energy Quantized Neural Networks*. 2017. arXiv: 1711.00215 [cs.NE].

[21] E. Park et al. "Big/little deep neural network for ultra low power inference". In: *2015 International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*. 2015, pp. 124–132. DOI: 10.1109/CODESISSS.2015.7331375.

[22] Josh Patterson and Adam Gibson. *Deep learning: a practitioners approach*. OReilly, 2017.

[23] Rafaelpadilla. *rafaelpadilla/Object-Detection-Metrics*. 2019. URL: https://github.com/rafaelpadilla/Object-Detection-Metrics.

[24] Mohammad Rastegari et al. *XNOR-Net: ImageNet Classification Using Binary Convolutional Neural Networks*. 2016. arXiv: 1603.05279 [cs.CV].

[25] Joseph Redmon and Ali Farhadi. "YOLO9000: Better, Faster, Stronger". In: *arXiv preprint arXiv:1612.08242* (2016).

[26] Joseph Redmon and Ali Farhadi. "YOLOv3: An Incremental Improvement". In: *arXiv* (2018).

[27] Joseph Redmon et al. "You Only Look Once: Unified, Real-Time Object Detection". In: *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)* (2016). DOI: 10.1109/cvpr.2016.91.

[28] Rajalingappaa Shanmugamani and Stephen Moore. *Deep learning for computer vision: expert techniques to train advanced neural networks using TensorFlow and Keras*. Packt., 2018.

[29] V. Sze et al. "Efficient Processing of Deep Neural Networks: A Tutorial and Survey". In: *Proceedings of the IEEE* 105.12 (2017), pp. 2295–2329. DOI: 10.1109/JPROC.2017.2761740.

[30] Vivienne Sze et al. "Efficient Processing of Deep Neural Networks: A Tutorial and Survey". In: *Proceedings of the IEEE* 105.12 (2017), 2295–2329. DOI: 10.1109/jproc.2017.2761740.

[31] Hokchhay Tann et al. "Runtime configurable deep neural networks for energy-accuracy trade-off". In: *Proceedings of the Eleventh IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis - CODES 16* (2016). DOI: 10.1145/2968456.2968458.

[32] *TensorFlow Core*. URL: https://www.tensorflow.org/guide.

[33] Bing Xu, Ruitong Huang, and Mu Li. *Revise Saturated Activation Functions*. 2016. arXiv: 1602.05980 [cs.LG].